Capítulo 6

VUELTA ATRÁS

6.1 INTRODUCCIÓN

Dentro de las técnicas de diseño de algoritmos, el método de Vuelta Atrás (del inglés *Backtracking*) es uno de los de más ámplia utilización, en el sentido de que puede aplicarse en la resolución de un gran número de problemas, muy especialmente en aquellos de optimización.

Los métodos estudiados en los capítulos anteriores construyen la solución basándose en ciertas propiedades de la misma; así en los algoritmos Ávidos se va contruyendo la solución por etapas, siempre avanzando sobre la solución parcial previamente calculada; o bien podremos utilizar la Programación Dinámica para dar una expresión recursiva de la solución si se verifica el principio de óptimo, y luego calcularla eficientemente. Sin embargo ciertos problemas no son susceptibles de solucionarse con ninguna de estas técnicas, de manera que la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posibles soluciones, en las que tratamos de encontrar una o todas las soluciones y por tanto también la óptima.

Para llevar a cabo este estudio exhaustivo, el diseño Vuelta Atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

En su forma básica la Vuelta Atrás se asemeja a un recorrido en profundidad dentro de un árbol cuya existencia sólo es implícita, y que denominaremos *árbol de expansión*. Este árbol es conceptual y sólo haremos uso de su organización como tal, en donde cada nodo de nivel *k* representa una parte de la solución y está formado por *k* etapas que se suponen ya realizadas. Sus hijos son las prolongaciones posibles al añadir una nueva etapa. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.

En este recorrido pueden suceder dos cosas. La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución (una hoja del árbol). Si lo único que buscabamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscabamos eran todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que llamamos *nodos fracaso*. En tal caso, el algoritmo vuelve atrás (y de ahí su

nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. En este retroceso, si existe uno o más caminos aún no explorados que puedan conducir a solución, el recorrido del árbol continúa por ellos.

La filosofía de estos algoritmos no sigue unas reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

Gran parte de la eficiencia (siempre relativa) de un algoritmo de Vuelta Atrás proviene de considerar el menor conjunto de nodos que puedan llegar a ser soluciones, aunque siempre asegurándonos de que el árbol "podado" siga conteniendo todas las soluciones. Por otra parte debemos tener cuidado a la hora de decidir el tipo de condiciones (*restricciones*) que comprobamos en cada nodo a fin de detectar nodos fracaso. Evidentemente el análisis de estas restricciones permite ahorrar tiempo, al delimitar el tamaño del árbol a explorar. Sin embargo esta evaluación requiere a su vez tiempo extra, de manera que aquellas restricciones que vayan a detectar pocos nodos fracaso no serán normalmente interesantes. No obstante, y como norma de actuación general, podríamos decir que las restricciones sencillas son siempre apropiadas, mientras que las más sofisticadas que requieren más tiempo en su cálculo deberían reservarse para situaciones en las que el árbol que se genera sea muy grande.

Vamos a ver como se lleva a cabo la búsqueda de soluciones trabajando sobre este árbol y su recorrido. En líneas generales, un problema puede resolverse con un algoritmo Vuelta Atrás cuando la solución puede expresarse como una n-tupla $[x_1, x_2, ..., x_n]$ donde cada una de las componentes x_i de este vector es elegida en cada etapa de entre un conjunto finito de valores. Cada etapa representará un nivel en el árbol de expansión.

En primer lugar debemos fijar la descomposición en etapas que vamos a realizar y definir, dependiendo del problema, la n-tupla que representa la solución del problema y el significado de sus componentes x_i. Una vez que veamos las posibles opciones de cada etapa quedará definida la estructura del árbol a recorrer. Vamos a ver a través de un ejemplo cómo es posible definir la estructura del árbol de expansión.

6.2 LAS *n* REINAS

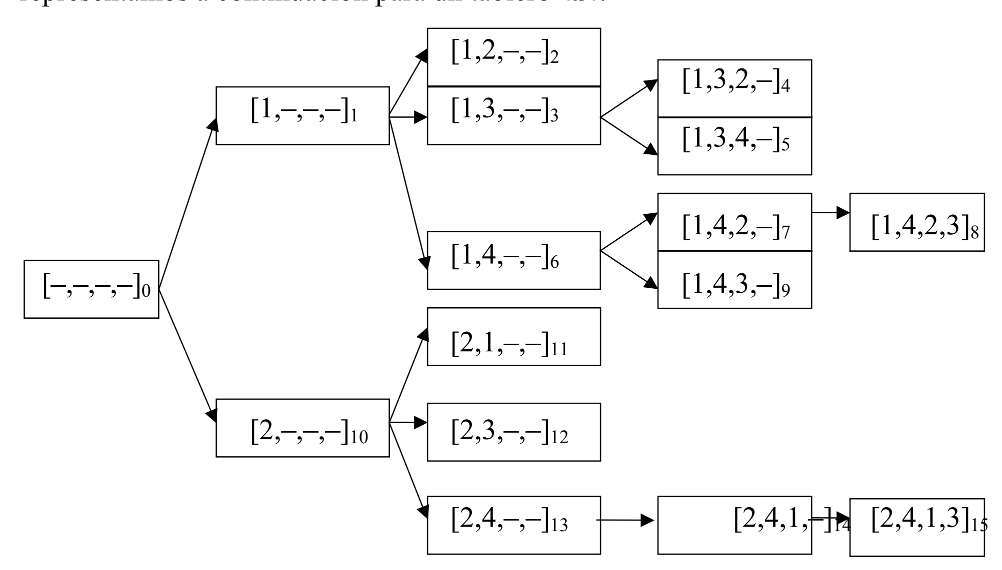
Un problema clásico que puede ser resuelto con un diseño Vuelta Atrás es el denominado de las ocho reinas y en general, de las n reinas. Disponemos de un tablero de ajedrez de tamaño 8x8, y se trata de colocar en él ocho reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren dos reinas ni en la misma fila, ni en la misma columna, ni en la misma diagonal.

Numeramos las reinas del 1 al 8. Cualquier solución a este problema estará representada por una 8-tupla $[x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8]$ en la que cada x_i representa la columna donde la reina de la fila i-ésima es colocada. Una posible solución al problema es la tupla [4,6,8,2,7,1,3,5].

Para decidir en cada etapa cuáles son los valores que puede tomar cada uno de los elementos x_i hemos de tener en cuenta lo que hemos denominado restricciones a fin de que el número de opciones en cada etapa sea el menor posible. En los algoritmos Vuelta Atrás podemos diferenciar dos tipos de restricciones:

- *Restricciones explícitas*. Formadas por reglas que restringen los valores que pueden tomar los elementos x_i a un conjunto determinado. En nuestro problema este conjunto es $S = \{1,2,3,4,5,6,7,8\}$.
- *Restricciones implícitas*. Indican la relación existente entre los posibles valores de los x_i para que éstos puedan formar parte de una *n*-tupla solución. En el problema que nos ocupa podemos definir dos restricciones implícitas. En primer lugar sabemos que dos reinas no pueden situarse en la misma columna y por tanto no puede haber dos x_i iguales (obsérvese además que la propia definición de la tupla impide situar a dos reinas en la misma fila, con lo cual tenemos cubiertos los dos casos, el de las filas y el de las columnas). Por otro lado sabemos que dos reinas no pueden estar en la misma diagonal, lo cual reduce el número de opciones. Esta condición se refleja en la segunda restricción implicita que, en forma de ecuación, puede ser expresada como $|x - x'| \neq |y - y'|$, siendo (x,y) y (x',y') las coordenadas de dos reinas en el tablero.

De esta manera, y aplicando las restricciones, en cada etapa *k* iremos generando sólo las *k*-tuplas con posibilidad de solución. A los prefijos de longitud *k* de la *n*-tupla solución que vamos construyendo y que verifiquen las restricciones expuestas los denominaremos *k-prometedores*, pues a priori pueden llevarnos a la solución buscada. Obsérvese que todo nodo generado es o bien fracaso o bien *k*-prometedor. Con estas condiciones queda definida la estructura del árbol de expansión, que representamos a continuación para un tablero 4x4:



Como podemos observar se construyen 15 nodos hasta dar con una solución al problema. El orden de generación de los nodos se indica con el subíndice que acompaña a cada tupla.

Conforme vamos construyendo el árbol debemos identificar los nodos que corresponden a posibles soluciones y cuáles por el contrario son sólo prefijos suyos. Ello será necesario para que, una vez alcanzados los nodos que sean posibles soluciones, comprobemos si de hecho lo son.

Por otra parte es posible que al alcanzar un cierto nodo del árbol sepamos que ninguna prolongación del prefijo de posible solución que representa va a ser solución a la postre (debido a las restricciones). En tal caso es absurdo que prosigamos buscando por ese camino, por lo que retrocederemos en el árbol (vuelta atrás) para seguir buscando por otra opción. Tales nodos son los que habíamos denominado nodos fracaso.

También es posible que aunque un nodo no se haya detectado a priori como fracaso (es decir, que sea *k*-prometedor) más adelante se vea que todos sus descendientes son nodos fracaso; en tal caso el proceso es el mismo que si lo hubiésemos detectado directamente. Tal es el caso para los nodos 2 y 3 de nuestro árbol. Efectivamente el nodo 2 es nodo fracaso porque al comprobar una de las restricciones (están en la misma diagonal) no se cumple. El nodo 3 sin embargo es nodo fracaso debido a que sus descendientes, los nodos 4 y 5, lo son.

Por otra parte hemos de identificar aquellos nodos que pudieran ser solución porque por ellos no se puede continuar (hemos completado la *n*-tupla), y aquellos que corresponden a soluciones parciales. No por conseguir construir un nodo hoja de nivel *n* quiere decir que hayamos encontrado una solución, puesto que para los nodos hojas también es preciso comprobar las restricciones. En nuestro árbol que representa el problema de las 4 reinas vemos cómo el nodo 8 podría ser solución ya que hemos conseguido colocar las 4 reinas en el tablero, pero sin embargo la tupla [1,4,2,3] encontrada no cumple el objetivo del problema, pues existen dos reinas $x_3 = 2$ y $x_4 = 3$ situadas en la misma diagonal. Un nodo con posibilidad de solución en el que detectamos que de hecho no lo es se comporta como nodo fracaso.

En resumen, podemos decir que Vuelta Atrás es un método exhaustivo de tanteo (prueba y error) que se caracteriza por un avance progresivo en la búsqueda de una solución mediante una serie de etapas. En dichas etapas se presentan unas opciones cuya validez ha de examinarse con objeto de seleccionar una de ellas para proseguir con el siguiente paso. Este comportamiento supone la generación de un árbol y su examen y eventual poda hasta llegar a una solución o a determinar su imposibilidad. Este avance se puede detener cuando se alcanza una solución, o bien si se llega a una situación en que ninguna de las soluciones es válida; en este caso se vuelve al paso anterior, lo que supone que deben recordarse las elecciones hechas en cada paso para poder probar otra opción aún no examinada. Este retroceso (vuelta atrás) puede continuar si no quedan opciones que examinar hasta llegar a la primera etapa. El agotamiento de todas las opciones de la primera etapa supondrá que no hay solución posible pues se habrán examinado todas las posibilidades.

El hecho de que la solución sea encontrada a través de ir añadiendo elementos a la solución parcial, y que el diseño Vuelta Atrás consista básicamente en recorrer un árbol hace que el uso de recursión sea muy apropiado. Los árboles son estructuras intrínsecamente recursivas, cuyo manejo requiere casi siempre de recursión, en especial en lo que se refiere a sus recorridos. Por tanto la

implementacion más sencilla se logra sin lugar a dudas con procedimientos recursivos.

De esta forma llegamos al esquema general que poseen los algoritmos que siguen la técnica de Vuelta Atrás:

```
PROCEDURE VueltaAtras(etapa);
BEGIN
   IniciarOpciones;
   REPEAT
      SeleccionarNuevaOpcion;
      IF Aceptable THEN
         AnotarOpcion;
         IF SolucionIncompleta THEN
            VueltaAtras(etapa_siguiente);
            IF NOT exito THEN
               CancelarAnotacion
            END
         ELSE (* solucion completa *)
            exito:=TRUE
         END
      END
   UNTIL (exito) OR (UltimaOpcion)
END VueltaAtras;
```

En este esquema podemos observar que están presentes tres elementos principales. En primer lugar hay una generación de descendientes, en donde para cada nodo generamos sus descendientes con posibilidad de solución. A este paso se le denomina expansión, ramificación o bifurcación. A continuación, y para cada uno de estos descendientes, hemos de aplicar lo que denominamos prueba de fracaso (segundo elemento). Finalmente, caso de que sea aceptable este nodo, aplicaremos la prueba de solución (tercer elemento) que comprueba si el nodo que es posible solución efectivamente lo es.

Tal vez lo más difícil de ver en este esquema es donde se realiza la vuelta atrás, y para ello hemos de pensar en la propia recursión y su mecanismo de funcionamiento, que es la que permite ir recorriendo el árbol en profundidad.

Para el ejemplo que nos ocupa, el de las *n* reinas, el algoritmo que lo soluciona quedaría por tanto como sigue:

```
CONST n = ...; (* numero de reinas; n>3 *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
VAR X:SOLUCION; exito:BOOLEAN;

PROCEDURE Reinas(k: CARDINAL);
```

```
(* encuentra una manera de disponer las n reinas *)
BEGIN
   IF k>n THEN RETURN END;
   X[k]:=0;
   REPEAT
      INC(X[k]); (* seleccion de nueva opcion *)
      IF Valido(k) THEN (* prueba de fracaso *)
         IF k<>n THEN
            Reinas(k+1) (* llamada recursiva *)
         ELSE
            exito:=TRUE
         END
      END
   UNTIL (X[k]=n) OR exito;
END Reinas;
```

La función *Valido* es la que comprueba las restricciones implícitas, realizando la prueba de fracaso:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
(* comprueba si el vector solucion X construido hasta el paso k
   es k-prometedor, es decir, si la reina puede situarse en la
   columna k *)
   VAR i: CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF (X[i]=X[k]) OR (ValAbs(X[i],X[k])=ValAbs(i,k)) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE
END Valido;
```

Utilizamos la funcion *ValAbs*(x,y), que es la que devuelve $|x - y|$:

```
PROCEDURE ValAbs(x,y:CARDINAL):CARDINAL;
BEGIN
   IF x>y THEN RETURN x-y ELSE RETURN y-x END;
END ValAbs;
```

Cuando se desea encontrar todas las soluciones habrá que alterar ligeramente el esquema dado, de forma que una vez conseguida una solución se continúe buscando hasta agotar todas las posibilidades. Queda por tanto el siguiente esquema general para este caso:

```
PROCEDURE VueltaAtrasTodasSoluciones(etapa);
```

```
BEGIN
   IniciarOpciones;
   REPEAT
      SeleccionarNuevaOpcion;
      IF Aceptable THEN
         AnotarOpcion;
         IF SolucionIncompleta THEN
            VueltaAtrasTodasSoluciones(etapa_siguiente);
         ELSE
            ComunicarSolucion
         END;
         CancelarAnotacion
      END
   UNTIL (UltimaOpcion);
END VueltaAtrasTodasSoluciones;
```

que en nuestro ejemplo de las reinas queda reflejado en el siguiente algoritmo:

```
PROCEDURE Reinas2(k:CARDINAL);
(* encuentra todas las maneras de disponer las n reinas *)
BEGIN
   IF k>n THEN RETURN END;
   X[k]:=0; (* iniciar opciones *)
   REPEAT
      INC(X[k]); (* seleccion de nueva opcion *)
      IF Valido(k) THEN (* prueba de fracaso *)
         IF k<>n THEN
            Reinas2(k+1) (* llamada recursiva *)
         ELSE
            ComunicarSolucion(X)
         END
      END
   UNTIL (X[k]=n);
END Reinas2;
```

Aunque la solución más utilizada es la recursión, ya que cada paso es una repetición del anterior en condiciones distintas (más simples), la resolución de este método puede hacerse también utilizando la organización del árbol que determina el espacio de soluciones. Así, podemos desarrollar también un esquema general que represente el comportamiento del algoritmo de Vuelta Atrás en su versión iterativa:

```
PROCEDURE VueltaAtrasIterativo;
BEGIN
```

```
  k:=1;
  WHILE k>1 DO
     IF solucion THEN
        ComunicarSolucion
     ELSIF Fracaso(solucion) OR (k<n) THEN
        DEC(k); CalcularSucesor(k)
     ELSE
        INC(k); CalcularSucesor(k)
     END
  END
END VueltaAtrasIterativo;
```

En este esquema también vemos presentes los tres elementos anteriores: prueba de solución, prueba de fracaso y generación de descendientes.

Para cada nodo se realiza la prueba de solución en cuyo caso se terminará el proceso y la prueba de fracaso que en caso positivo da lugar a la vuelta atrás. Observamos también que si la búsqueda de descendientes no consigue ningún hijo, el nodo se convierte en nodo fracaso y se trata como en el caso anterior; en caso contrario la etapa se incrementa en uno y se continúa.

Por otra parte la vuelta atrás busca siempre un hermano del nodo que estemos analizando –descendiente de su mismo padre– para pasar a su análisis; si no existe tal hermano se decrementa la etapa *k* en curso y si *k* sigue siendo mayor que cero (aun no hemos recorrido el árbol) se repite el proceso anterior.

El algoritmo iterativo para el problema de las *n* reinas puede implementarse por tanto utilizando este esquema, lo que da lugar al siguiente procedimiento:

```
PROCEDURE Reinas_It;
   VAR k:CARDINAL;
BEGIN
   X[1]:=0; k:=1;
   WHILE k>0 DO
      X[k]:=X[k] + 1; (* selecciona nueva opcion *)
      WHILE (X[k]<=n)AND(NOT Valido(k)) DO (* fracaso? *)
         X[k]:=X[k] + 1
      END
      IF X[k]<=n THEN
         IF k=n THEN   ComunicarSolucion(X)
         ELSE INC(k); X[k]:=0
         END
      ELSE
         DEC(k) (* vuelta atras *)
      END
   END
END Reinas_It;
```

Hemos visto en este apartado cómo generar el árbol de expansión, pero sin prestar demasiada atención al orden en que lo hacemos. Usualmente los algoritmos

Vuelta Atrás son de complejidad exponencial por la forma en la que se busca la solución mediante el recorrido en profundidad del árbol. De esta forma estos algoritmos van a ser de un orden de complejidad al menos del número de nodos del árbol que se generen y este número, si no se utilizan restricciones, es de orden de z^n donde z son las posibles opciones que existen en cada etapa, y n el número de etapas que es necesario recorrer hasta construir la solución (esto es, la profundidad del árbol o la longitud de la n-tupla solución).

El uso de restricciones, tanto implícitas como explícitas, trata de reducir este número tanto como sea posible (en el ejemplo de las reinas se pasa de 8^8 nodos si no se usa ninguna restricción a poco más de 2000), pero sin embargo en muchos casos no son suficientes para conseguir algoritmos "tratables", es decir, que sus tiempos de ejecución sean de orden de complejidad razonable.

Para aquellos problemas en donde se busca una solución y no todas, es donde entra en juego la posibilidad de considerar distintas formas de ir generando los nodos del árbol. Y como la búsqueda que realiza la Vuelta Atrás es siempre en profundidad, para lograr esto sólo hemos de ir variando el orden en el que se generan los descendientes de un nodo, de manera que trate de ser lo más apropiado a nuestra estrategia.

Como ejemplo, pensemos en el problema del laberinto que veremos más adelante. En cada etapa vamos a ir generando los posibles movimientos desde la casilla en la que nos encontramos. Pero en vez de hacerlo de cualquier forma, sería interesante explorar primero aquellos que nos puedan llevar más cerca de la casilla de salida, es decir, tratar de ir siempre hacia ella.

Desde un punto de vista intuitivo, lo que intentamos hacer así es llevar lo más hacia arriba posible del árbol de expansión el nodo hoja con la solución (dibujando el árbol con la raiz a la izquierda, igual que lo hemos hecho en el problema de las reinas), para que la búsqueda en profundidad que realizan este tipo de algoritmos la encuentre antes. En algunos ejemplos, como puede ser en el del juego del Continental, que también veremos más adelante, el orden en el que se generan los movimientos hace que el tiempo de ejecución del algoritmo pase de varias horas a sólo unos segundos, lo cual no es despreciable.

6.3 RECORRIDOS DEL REY DE AJEDREZ

Dado un tablero de ajedrez de tamaño $n \times n$, un rey es colocado en una casilla arbitraria de coordenadas (x,y). El problema consiste en determinar los n^2-1 movimientos de la figura de forma que todas las casillas del tablero sean visitadas una sola vez, si tal secuencia de movimientos existe.

Solución (☺)

La solución al problema puede expresarse como una matriz de dimensión $n \times n$ que representa el tablero de ajedrez. Cada elemento (x,y) de esta matriz solución contendrá un número natural k que indica el número de orden en que ha sido visitada la casilla de coordenadas (x,y).

El algoritmo trabaja por etapas decidiendo en cada etapa k hacia donde se mueve. Como existen ocho posibles movimientos en cada etapa, éste será el número máximos de hijos que se generarán por cada nodo.

Respecto a las restricciones explícitas, por la forma en la que hemos definido la estructura que representa la solución (en este caso una matriz bidimensional de números naturales), sabemos que sus componentes pueden ser números comprendidos entre cero (que indica que una casilla no ha sido visitada aún) y n^2, que es el orden del último movimiento posible. Inicialmente el tablero se encuentra relleno con ceros y sólo existe un 1 en la casilla inicial (x_0,y_0).

Las restricciones implícitas en este caso van a limitar el número de hijos que se generan desde una casilla mediante la comprobación de que el movimiento no lleve al rey fuera del tablero o sobre una casilla previamente visitada.

Una vez definida la estructura que representa la solución y las restricciones que usaremos, para implementar el algoritmo que resuelve el problema basta utilizar el esquema general, obteniendo:

```
CONST n = ...;
TYPE TABLERO = ARRAY[1..n],[1..n] OF CARDINAL;
VAR tablero:TABLERO;

PROCEDURE Rey1(k:CARDINAL;x,y:INTEGER;VAR exito:BOOLEAN);
   (* busca una solucion, si la hay. k indica la etapa, (x,y) las
      coordenadas de la casilla en donde se encuentra el rey *)
   VAR orden:CARDINAL;(* recorre cada uno de los 8 movimientos *)
       u,v:INTEGER; (* u,v indican la casilla destino desde x,y *)
BEGIN
   orden:=0;
   exito:=FALSE;
   REPEAT
      INC(orden);
      u:= x + mov_x[orden];
      v:= y + mov_y[orden];
      IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
         tablero[u,v]:= k;
         IF k<n*n THEN
            Rey1(k+1,u,v,exito);
            IF NOT exito THEN tablero[u,v]:=0 END
         ELSE exito:= TRUE;
         END
      END
   UNTIL (exito) OR (orden=8);
END Rey1;
```

Las variables *mov_x* y *mov_y* contienen los movimientos legales de un rey (según las reglas de ajedrez), y son inicializadas al principio del programa principal mediante el procedimiento *MovimientosPosibles*:

```
VAR mov_x,mov_y:ARRAY[1..8] OF INTEGER;
```

```
PROCEDURE MovimientosPosibles;
BEGIN
   mov_x[1]:=0;  mov_y[1]:=1;  mov_x[2]:=-1; mov_y[2]:=1;
   mov_x[3]:=-1; mov_y[3]:=0;  mov_x[4]:=-1; mov_y[4]:=-1;
   mov_x[5]:=0;  mov_y[5]:=-1; mov_x[6]:=1; mov_y[6]:=-1;
   mov_x[7]:=1; mov_y[7]:=0;   mov_x[8]:=1; mov_y[8]:=1;
END MovimientosPosibles;
```

El programa principal es también el encargado de inicializar el tablero e invocar al procedimiento *Rey1* con los parámetros iniciales:

```
....
MovimientosPosibles();
FOR i:=1 TO n DO
   FOR j:=1 TO n DO
      tablero[i,j]:=0;
   END
END;
tablero[x0,y0]:=1; (* x0,y0 es la casilla inicial *)
Rey1(2,x0,y0,exito);
....
```

Supongamos que nos piden ahora una modificación al programa de forma que, en vez de encontrar los movimientos, calcule cuántas soluciones posee el problema, es decir, cuántos recorridos válidos distintos puede hacer el rey desde la casilla inicial dada. En este caso utilizaremos el esquema que permite encontrar todas las soluciones, lo que da lugar al siguiente programa:

```
PROCEDURE Rey2(k:CARDINAL;x,y:INTEGER;
               VAR numsoluciones:CARDINAL);
(* cuenta todas las soluciones *)
   VAR orden:CARDINAL;(*recorre cada uno de los 8 movimientos *)
       u,v:INTEGER; (* u,v indican la casilla destino desde x,y *)
BEGIN
   orden:=0;
   REPEAT
      INC(orden);
      u:= x + mov_x[orden];
      v:= y + mov_y[orden];
      IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
         tablero[u,v]:= k;
         IF k<n*n THEN
            Rey2(k+1,u,v,numsoluciones)
         ELSE
```

```
          INC(numsoluciones)
        END;
        tablero[u,v]:=0
      END
    UNTIL (orden=8);
  END Rey2;
```

6.4 RECORRIDOS DEL REY DE AJEDREZ (2)

Al igual que en el problema discutido anteriormente, un rey es colocado en una casilla arbitraria de coordenadas (x_0,y_0) de un tablero de ajedrez de tamaño nxn.

Si asignamos a cada casilla del tablero un peso (dado por el producto de sus coordenadas), a cada posible recorrido le podemos asignar un valor que viene dado por la suma de los pesos de las casillas visitadas por el índice del movimiento que nos llevó a esa casilla dentro del recorrido.

Esto es, si (x_0,y_0) es la casilla inicial y el recorrido R viene dado por los movimientos [(x_1,y_1), (x_2,y_2), ..., (x_k,y_k)], con $k = n^2 – 1$, el peso asignado a R vendrá dado por la expresión:

$$P(R) = \sum_{i=0}^{k} i x_i y_i \, .$$

El problema consiste en averiguar el recorrido de peso mínimo para una casilla inicial dada.

Solución (☺)

Utilizaremos las mismas estructuras de datos que en el problema anterior, al igual que las restricciones. La modificación pedida es simple, pues es una pequeña variación del procedimiento *Rey2* que va recorriendo el árbol de expansión contando el número de soluciones:

```
PROCEDURE Rey3(k:CARDINAL;x,y:INTEGER);
  VAR orden,costerecorrido:CARDINAL; u,v:INTEGER;
BEGIN
  orden:=0;
  REPEAT
    INC(orden);
    u:= x + mov_x[orden]; v:= y + mov_y[orden];
    IF (1<=u)AND(u<=n)AND(1<=v)AND(v<=n)AND(tablero[u,v]=0) THEN
      tablero[u,v]:= k;
      IF k<n*n THEN
        Rey3(k+1,u,v)
      ELSE
        costerecorrido:=CalcularCoste(tablero);
        IF costerecorrido<costeminimo THEN
```

```
                costeminimo:=costerecorrido;
                mejorsolucion:=tablero
              END
            END;
            tablero[u,v]:=0
          END
        UNTIL (orden=8);
      END Rey3;
```

En este ejemplo utilizamos las variables globales *costeminimo* y *mejorsolucion*:

```
VAR costeminimo:CARDINAL; mejorsolucion:TABLERO;
```

que almacenan la mejor solución encontrada hasta el momento. La primera será inicializada en el cuerpo principal del programa, y la segunda de ellas no hace falta inicializar:

```
...
costeminimo:=MAX(CARDINAL);
MovimientosPosibles();
FOR i:=1 TO n DO
   FOR j:=1 TO n DO
      tablero[i,j]:=0
   END
END;
tablero[x0,y0]:=1; (* x0,y0 es la casilla inicial *)
Rey3(2,x0,y0);
...
```

Por su parte, la función *CalcularCoste* es la que determina el peso de un recorrido dado:

```
PROCEDURE CalcularCoste(VAR t:TABLERO):CARDINAL;
   VAR i,j,coste:CARDINAL;
BEGIN
   coste:=0;
   FOR i:=1 TO n DO
      FOR j:=1 TO n DO
         coste:=coste+t[i,j]*i*j
      END
   END;
   RETURN coste;
END CalcularCoste;
```

Existe una pequeña variación a este algoritmo que consiste en ir acarreando el coste del recorrido en curso a lo largo del recorrido del árbol, lo que supondría ahorrarse la llamada (de orden O(n^2)) a la función *CalcularCoste* cada vez que se

alcance una hoja del árbol de expansión (esto es, una posible solución). Para implementarlo, basta con incluir como un parámetro más del procedimiento *Rey3* una variable que lleve acumulado el peso del recorrido hasta el momento. Cada vez que se escoja una nueva opción se incrementará tal valor, y cada vez que se cancele una anotación se decrementará en las unidades correspondientes.

Por otro lado, también es posible plantearse si existe un algoritmo más eficiente que el de Vuelta Atrás para resolver este problema. Observando la forma en la que se define la función de peso, el algoritmo debe tratar siempre de visitar las casillas de mayores coordenadas primero, para que el número de orden en que son visitadas, que es un factor multiplicativo en la función, sea lo menor posible.

Así, nos encontramos delante de un típico algoritmo ávido, que escogería siempre como siguiente casilla (x,y) a mover de entre las posibles a aquella aún no visitada y cuyo producto de coordenadas xy sea máximo. La complejidad de este algoritmo es de orden $O(n^2)$, mucho más eficiente que el de Vuelta Atrás. Sin embargo, la demostración de que siempre encuentra la solución no es sencilla.

6.5 LAS PAREJAS ESTABLES

Supongamos que tenemos n hombres y n mujeres y dos matrices M y H que contienen las preferencias de los unos por los otros. Más concretamente, la fila $M[i,\cdot]$ es una ordenación (de mayor a menor) de las mujeres según las preferencias del i-ésimo hombre y, análogamente, la fila $H[i,\cdot]$ es una ordenación (de mayor a menor) de los hombres según las preferencias de la i-ésima mujer.

El problema consiste en diseñar un algoritmo que encuentre, si es que existe, un emparejamiento de hombres y mujeres tal que todas las parejas formadas sean *estables*. Diremos que una pareja (h,m) es estable si no se da ninguna de estas dos circunstancias:

1) Existe una mujer m' (que forma la pareja (h',m')) tal que el hombre h la prefiere sobre la mujer m y además la mujer m' también prefiere a h sobre h'.
2) Existe un hombre h'' (que forma la pareja (h'',m'')) tal que la mujer m lo prefiere sobre el hombre h y además el hombre h'' también prefiere a m sobre la mujer m''.

Solución (☺)

Para este problema vamos a disponer de una n-tupla X que vamos a ir rellenando en cada etapa del algoritmo, y que contiene las mujeres asignadas a cada uno de los hombres. En otras palabras, x_i indicará el número de la mujer asignada al i-ésimo hombre en el emparejamiento. El algoritmo que resuelve el problema trabajará por etapas y en cada etapa k decide la mujer que ha de emparejarse con el hombre k.

Analicemos en primer lugar las restricciones del problema. En una etapa cualquiera k, el k-ésimo hombre escogerá la mujer que prefiere en primer lugar, siempre y cuando esta mujer aún esté libre y la pareja resulte estable. Para saber las mujeres aún libres utilizaremos un vector auxiliar denominado *libre*. Por simetría, aparte de la n-tupla X, también dispondremos de otra n-tupla Y que contiene los hombres asignados a cada mujer, que necesitaremos al comprobar las restricciones.

Por último, también son necesarias dos tablas auxiliares, *ordenM* y *ordenH*. La primera almacena en la posición [*i*,*j*] el orden de preferencia de la mujer *i* por el hombre *j*, y la segunda almacena en la posición [*i*,*j*] el orden de preferencia del hombre *i* por la mujer *j*.

Con todo esto, el procedimiento que resuelve el problema puede ser implementado como sigue:

```
TYPE PREFERENCIAS = ARRAY [1..n],[1,,n] OF CARDINAL;
     ORDEN = ARRAY [1..n],[1,,n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;
     DISPONIBILIDAD = ARRAY [1..n] OF BOOLEAN;

VAR  M,H:PREFERENCIAS;
     ordenM,ordenH:ORDEN;
     X,Y:SOLUCION;
     libre:DISPONIBILIDAD;

PROCEDURE Parejas(hombre:CARDINAL;VAR exito:BOOLEAN);
   VAR mujer,prefiere,preferencias:CARDINAL;
BEGIN
   prefiere:=0; (* recorre las posibles elecciones del hombre *)
   REPEAT
      INC(prefiere);
      mujer:=M[hombre,prefiere];
      IF libre[mujer] AND Estable(hombre,mujer,prefiere) THEN
         X[hombre]:=mujer;
         Y[mujer]:=hombre;
         libre[mujer]:=FALSE;
         IF hombre<n THEN
            Parejas(hombre+1,exito);
            IF NOT exito THEN
               libre[mujer]:=TRUE
            END
         ELSE
            exito:=TRUE;
         END
      END
   UNTIL (prefiere=n) OR exito;
END Parejas;
```

La función *Estable* queda definida como:

```
PROCEDURE Estable(h,m,p:CARDINAL):BOOLEAN;
```

```
   VAR mejormujer,mejorhombre,i,limite:CARDINAL;s:BOOLEAN;
BEGIN
   s:=TRUE; i:=1;
   WHILE (i<p) AND s DO (* es estable respecto al hombre? *)
      mejormujer:=M[h,i];
      INC(i);
      IF NOT(libre[mejormujer])THEN
         s:=ordenM[mejormujer,h]>ordenM[mejormujer,Y[mejormujer]];
      END
   END;
   i:=1; limite:=H[m,h]; (* es estable respecto a la mujer? *)
   WHILE(i<limite) AND s DO
      mejorhombre:=H[m,i];
      INC(i);
      IF mejorhombre<h THEN
      s:=ordenH[mejorhombre,m]>ordenH[mejorhombre,X[mejorhombre]];
      END
   END;
   RETURN s
END Estable;
```

El problema se resuelve mediante la inicialización apropiada de las matrices de preferencias y una invocación a *Parejas*(1,*exito*). Tras su ejecución, las variables *X* e *Y* contendrán la asignaciones respectivas siempre que la variable *exito* lo indique.

6.6 EL LABERINTO

Una matriz bidimensional $n \times n$ puede representar un laberinto cuadrado. Cada posición contiene un entero no negativo que indica si la casilla es transitable (0) o no lo es (∞). Las casillas [1,1] y [n,n] corresponden a la entrada y salida del laberinto y siempre serán transitables.

Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida.

Solución (☺)

En este problema iremos avanzando por el laberinto en cada etapa, y cada nodo representará el camino recorrido hasta el momento. Por la forma en la que trabaja el esquema general de Vuelta Atrás podemos utilizar una variable global (una matriz) para representar el laberinto e ir apuntando los movimientos que realizamos, indicando en cada casilla el orden en el que ésta ha sido visitada. Al producirse la vuelta atrás nos cuidaremos de liberar las casillas ocupadas por el nodo del que volvemos (marcándolas de nuevo con 0).

Con esto en mente, el algoritmo es sencillo:

```
CONST n = ...;
```

```
TYPE LABERINTO = ARRAY[1..n],[1..n] OF CARDINAL;
VAR lab:LABERINTO;

PROCEDURE Laberinto(k:CARDINAL;VAR fil,col:INTEGER;
                    VAR exito:BOOLEAN);
   VAR orden:CARDINAL; (*indica hacia donde debe moverse *)
BEGIN
   orden:=0; exito:=FALSE;
   REPEAT
      INC(orden);
      fil:=fil + mov_fil[orden];
      col:=col + mov_col[orden];
      IF (1<=fil) AND (fil<=n) AND (1<=col) AND (col<=n) AND
      (lab[fil,col]=0) THEN
         lab[fil,col]:=k;
         IF (fil=n) AND (col=n) THEN exito:=TRUE
         ELSE
            Laberinto(k+1,fil,col,exito);
            IF NOT exito THEN lab[fil,col]:=0 END
         END
      END;
      fil:=fil - mov_fil[orden];
      col:=col - mov_col[orden]
      END
   UNTIL (exito) OR (orden=4)
END Laberinto;
```

Las variables *mov_fil* y *mov_col* contienen los posibles movimientos, y son inicializadas por el procedimiento *MovimientosPosibles* que mostramos a continuación:

```
VAR mov_fil,mov_col:ARRAY [1..4] OF INTEGER;

PROCEDURE MovimientosPosibles;
BEGIN
   mov_fil[1]:=1;  mov_col[1]:=0;  (* sur *)
   mov_fil[2]:=0;  mov_col[2]:=1;  (* este *)
   mov_fil[3]:=0;  mov_col[3]:=-1; (* oeste *)
   mov_fil[4]:=-1; mov_col[4]:=0;  (* norte *)
END MovimientosPosibles;
```

Tal como mencionamos en la introducción de este capítulo, el orden en que se intentan esos movimientos es importante, pues podemos utilizar la información disponible de que la casilla de salida es la [*n*,*n*] para tratar siempre de ir hacia ella. Ésta es la razón por la que se intenta primero el sur, luego el este, y por último el oeste y el norte. Para cambiar la forma en la que se realizan los intentos, y por tanto

el orden en el que se construyen los nodos del árbol de expansión, basta modificar el procedimiento *MovimientosPosibles*, que es invocado una vez al comienzo del programa principal –que también es el que invoca la primera vez a la función *Laberinto*–. Por supuesto, independientemente de esta ordenación, el algoritmo encuentra la salida si es que ésta es alcanzable.

Mayor importancia tendría este orden de expandir los nodos si lo que se desease fuera encontrar no una solución cualquiera, sino la más corta. Para ello el algoritmo que lo realiza está basado en el esquema general que busca todas las soluciones y se queda con la mejor:

```
PROCEDURE Laberinto2(k:CARDINAL;VAR fil,col:INTEGER);
   VAR orden:CARDINAL; (*indica hacia donde debe moverse *)
BEGIN
   orden:=0;
   REPEAT
      INC(orden);
      fil:=fil + mov_fil[orden];
      col:=col + mov_col[orden];
      IF (1<=fil) AND (fil<=n) AND (1<=col) AND (col<=n) AND
         (lab[fil,col]=0) THEN
         lab[fil,col]:=k;
         IF (fil=n) AND (col=n) THEN
            (* almacenamos el mejor recorrido hasta el momento *)
            recorridominimo:=k; solucion:=lab;
         ELSIF k<=recorridominimo THEN (* poda! *)
            Laberinto2(k+1,fil,col);
         END;
         lab[fil,col]:=0;
      END;
      fil:=fil - mov_fil[orden];
      col:=col - mov_col[orden]
   UNTIL (orden=4)
END Laberinto2;
```

En este caso hemos introducido una variante muy importante: el uso de una cota para podar ramas del árbol de expansión. Si bien ésta es una técnica que se utiliza sobre todo en los algoritmos de Ramificación y Poda (y de ahí su nombre), el uso de cotas para podar puede ser aplicado a cualquier tipo de árboles de expansión.

En el problema que nos ocupa calculamos la primera solución y para ella se obtiene un valor. En este caso es el número de movimientos que ha realizado el algoritmo hasta encontrar la salida, que es el valor que queremos minimizar. Pues bien, disponiendo ya de un valor alcanzable podemos "rechazar" todos aquellos nodos cuyos recorridos superen este valor, sean soluciones parciales o totales, pues no nos van a llevar hacia la solución óptima. Estas *podas* ahorran mucho trabajo al algoritmo, pues evitan que éste realice trabajo innecesario.

La variable *recorridominimo*, que es la que hace las funciones de cota, se inicializa a *MAX(CARDINAL)* al principio del programa principal que invoca al procedimiento *Laberinto2*(1,1,1).

Como norma general para los algoritmos de Vuelta Atrás, y puesto que su complejidad es normalmente exponencial, debemos de saber aprovechar toda la información disponible sobre el problema o sus soluciones en forma de restricciones, pues son éstas la clave de su eficiencia. En la mayoría de los casos la diferencia entre un algoritmo Vuelta Atrás útil y otro que, por su tardanza, no pueda utilizarse se encuentra en las restricciones impuestas a los nodos, único parámetro disponible al programador de estos métodos para mejorar la eficiencia de los algoritmos resultantes.

6.7 LA ASIGNACIÓN DE TAREAS

Este problema fue introducido en el capítulo cuatro (apartado 4.11), y básicamente puede ser planteado como sigue. Dadas n personas y n tareas, queremos asignar a cada persona una tarea. El coste de asignar a la persona i la tarea j viene determinado por la posición $[i,j]$ de una matriz dada (TARIFAS). Diseñar un algoritmo que asigne una tarea a cada persona minimizando el coste de la asignación.

Solución (☺)

En primer lugar hemos de decidir cómo representaremos la solución del problema mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$. En este ejemplo el valor x_i va a representar la tarea asignada a la i-ésima persona. Empezando por la primera persona, en cada etapa el algoritmo irá avanzando en la construcción de la solución, comprobando siempre que el nuevo valor añadido a ella es compatible con los valores anteriores. Por cada solución que encuentre anotará su coste y lo comparará con el coste de la mejor solución encontrada hasta el momento, que almacenará en la variable global *mejor*. El algoritmo puede ser implementado como sigue:

```
CONST n = ...; (* numero de personas y trabajos *)
TYPE  TARIFAS = ARRAY[1..n],[1..n] OF CARDINAL;
      SOLUCION = ARRAY[1..n] OF CARDINAL;

VAR   X,mejor:SOLUCION;
      minimo:CARDINAL;
      tarifa:TARIFAS;

PROCEDURE Asignacion(k:CARDINAL);
   VAR c:CARDINAL;
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
```

```
      IF Aceptable(k) THEN
         IF k<n THEN Asignacion(k+1)
         ELSE
            c:=Coste();
            IF minimo>c THEN mejor:=X; minimo:=c END;
         END
      END
   UNTIL X[k]=n
END Asignacion;
```

Siendo los procedimientos *Aceptable* y *Coste* los que respectivamente comprueban las restricciones para este problema y calculan el coste de las soluciones que van generándose. En cuanto a las restricciones, sólo se va a definir una, que asegura que las tareas sólo se asignan una vez. Por otro lado, el coste de una solución coincide con el coste de la asignación:

```
PROCEDURE Aceptable(k:CARDINAL):BOOLEAN;
(* comprueba que esa tarea no ha sido asignada antes *)
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF X[k]= X[i] THEN RETURN FALSE END
   END;
   RETURN TRUE
END Aceptable;


PROCEDURE Coste():CARDINAL;
   VAR suma,i:CARDINAL;
BEGIN
   suma:=0;
   FOR i:=1 TO n DO
      suma:=suma+tarifa[i,X[i]]
   END;
   RETURN suma
END Coste;
```

Para resolver el problema basta con invocar al procedimiento *Asignación* tras obtener los valores de la tabla de tarifas e inicializar la variable *minimo*:

```
...
minimo:=MAX(CARDINAL);
Asignacion(1);
ComunicarSolucion(mejor);
...
```

Al igual que en el problema anterior, existe una modificación a este algoritmo que permite realizar podas al árbol de expansión eliminando aquellos nodos que no van a llevar a la solución óptima. Para implementar esta modificación –siempre interesante puesto que consigue reducir el tamaño del árbol de búsqueda– necesitamos hacer uso de una cota que almacene el valor obtenido por la mejor solución hasta el momento, y además llevar la cuenta en cada nodo del coste acumulado hasta ese nodo. Si el coste acumulado es mayor que el valor de la cota, no merece la pena continuar explorando los hijos de ese nodo, pues nunca nos llevarán a una solución mejor de la que teníamos.

Como la cota la tenemos disponible ya (es la variable *minimo* del algoritmo anterior), lo que haremos es ir calculando de forma acumulada en vez de al llegar a una solución, y así podremos realizar la poda cuanto antes:

```
PROCEDURE Asignacion2(k:CARDINAL;costeacum:CARDINAL);
   VAR coste:CARDINAL;
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      coste:=costeacum+tarifa[k,X[k]];
      IF Aceptable(k) AND (coste<=minimo) THEN
         IF k<n THEN Asignacion2(k+1,coste)
         ELSE mejor:=X; minimo:=coste
         END
      END
   UNTIL X[k]=n
END Asignacion2;
```

También hacer una última observación sobre el problema de la asignación en general. Este problema siempre posee solución puesto que siempre existen asignaciones válidas. De esta forma no nos tenemos que preocupar de si el algoritmo acaba con éxito o no.

6.8 LA MOCHILA (0,1)

El problema de la mochila (0,1) –originalmente descrito en el apartado 4.8– ha sido discutido en los dos últimos capítulos, y hemos visto que no posee solución mediante un algoritmo ávido, aunque sí la tiene utilizado Programación Dinámica. Nos planteamos aquí dar una solución al problema utilizando Vuelta Atrás.

Recordemos el enunciado del problema. Dados n elementos e_1, e_2, ..., e_n con pesos p_1, p_2, ..., p_n y beneficios b_1, b_2, ..., b_n, y dada una mochila capaz de albergar hasta un máximo peso M (capacidad de la mochila), queremos encontrar cuáles de los n elementos hemos de introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima, sujeto a la restricción de que tales elementos no pueden superar la capacidad de la mochila.

Solución (☺/✍)

Éste es uno de los problemas cuya resolución más sencilla se obtiene utilizando la técnica de Vuelta Atrás, puesto que basta expresar la solución al problema como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde x_i tomará los valores 1 ó 0 dependiendo de que el i-ésimo objeto sea introducido o no. El árbol de expansión resultante es, por tanto, trivial.

Sin embargo, y puesto que se trata de un problema de optimización, podemos aplicar una poda para eliminar aquellos nodos que no conduzcan a una solución óptima. Para ello vamos utilizaremos una función (*Cota*) que describiremos más adelante.

Como su versión recursiva no plantea mayores dificultades por tratarse de un mero ejercicio de aplicación del esquema general, este problema lo resolveremos mediante un algoritmo iterativo:

```
CONST n = ...; (* numero de elementos *)
      M = ...; (* capacidad de la mochila *)

TYPE REGISTRO = RECORD peso,beneficio:REAL END;
      ELEMENTOS = ARRAY[1..n] OF REGISTRO;
      MOCHILA   = ARRAY[1..n] OF CARDINAL;

PROCEDURE Mochila(elem:ELEMENTOS;capacidad:REAL;VAR X:MOCHILA;
                        VAR peso_final,beneficio_final:REAL);
   VAR peso_en_curso,beneficio_en_curso:REAL;
       sol:MOCHILA; (* solucion en curso *)
       k:CARDINAL;
BEGIN
   peso_en_curso:=0.0;
   beneficio_en_curso:=0.0;
   beneficio_final:=-1.0;
   k:=1;
   LOOP
      WHILE  (k<=n) AND (peso_en_curso+elem[k].peso<=capacidad) DO
         peso_en_curso:=peso_en_curso+elem[k].peso;
         beneficio_en_curso:=beneficio_en_curso+elem[k].beneficio;
         sol[k]:=1;
         INC(k)
      END;
      IF k>n THEN
         beneficio_final:=beneficio_en_curso;
         peso_final:=peso_en_curso;
         k:=n;
         X:=sol;
      ELSE
         sol[k]:=0
      END;
```

```
      WHILE Cota(elem,beneficio_en_curso,peso_en_curso,k,capacidad)<=
                beneficio_final DO
         WHILE (k<>0) AND (sol[k]<>1) DO DEC(k) END;
         IF k=0 THEN  EXIT  END;
         sol[k]:=0;
         peso_en_curso:=peso_en_curso-elem[k].peso;
         beneficio_en_curso:=beneficio_en_curso-elem[k].beneficio
      END;
      INC(k)
   END
END Mochila;
```

La función *Cota* es la que va a permitir realizar la poda del árbol de expansión para aquellos nodos que no lleven a la solución óptima. Para ello vamos a considerar que los elementos iniciales están todos ordenados de forma decreciente por su ratio beneficio/peso. De esta forma, supongamos que nos encontramos en el paso *k*-ésimo, y que disponemos de un beneficio acumulado B_k. Por la manera en como hemos ido construyendo el vector, sabemos que

$$B_k = \sum_{i=1}^{k} sol[i] * elem[i].beneficio\ .$$

Para calcular el valor máximo que podríamos alcanzar con ese nodo (B_M), vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de ratio beneficio/peso, este mejor elemento será el siguiente (k+1). Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos "aspirar" si seguimos por esa rama del árbol:

$$B_M = B_k + \left(capacidad - \sum_{i=1}^{k} sol[i] * elem[i].peso \right) \frac{elem[k+1].beneficio}{elem[k+1].peso}$$

Con esto:

```
PROCEDURE Cota(e:ELEMENTOS;b:REAL;p:REAL;k:CARDINAL;cap:CARDINAL):REAL;
   VAR i: CARDINAL; ben_ac,peso_ac:REAL; (* acumulados *)
BEGIN
   ben_ac:=b; peso_ac:=p;
   FOR i:=k+1 TO n DO
      peso_ac:=peso_ac+e[i].peso;
      IF peso_ac<cap THEN ben_ac:=ben_ac+e[i].beneficio
      ELSE
         RETURN (ben_ac+(1.0-(peso_ac-cap)/e[i].peso)*(e[i].beneficio))
      END
   END;
   RETURN (beneficio_acumulado)
END Cota;
```

El tipo de poda que hemos realizado al árbol de expansión de este ejercicio está más cerca de las técnicas de poda que se utilizan en los algoritmos de Ramificación y Poda que de las típicas restricciones definidas para los algoritmos Vuelta Atrás.

Aparte de las diferencias existentes entre ambos tipos de algoritmos en cuanto a la forma de recorrer el árbol (mucho más flexible en la técnica de Ramificación y Poda) y la utilización de estructuras globales en Vuelta Atrás (frente a los nodos "autónomos" de la Ramificación y Poda), no existe una frontera clara entre los procesos de poda de unos y otros. En cualquier caso, repetimos la importancia de la poda, esencial para convertir en tratables estos algoritmos de orden exponencial.

6.9 LOS SUBCONJUNTOS DE SUMA DADA

Sea W un conjunto de enteros no negativos y M un número entero positivo. El problema consiste en diseñar un algoritmo para encontrar todos los posibles subconjuntos de W cuya suma sea exactamente M.

Solución (☺)

En primer lugar podemos suponer sin pérdida de generalidad (ni de eficiencia, porque a la postre el algoritmo resultante es de complejidad exponencial) que el conjunto viene dado por un vector de enteros no negativos y que ya se encuentra ordenado de forma creciente. Con esto en mente, la solución al problema podremos expresarla como una n-tupla $X = [x_1, x_2, ..., x_n]$ en donde x_i podrá tomar los valores 1 ó 0 indicando que el entero i forma parte de la solución o no. El algoritmo trabaja por etapas y en cada etapa ha de decidir si el k-ésimo entero del conjunto interviene o no en la solución.

A modo de ejemplo, supongamos el conjunto $W = \{2,3,5,10,20\}$ y sea M=15. Existen dos posibles soluciones, dadas por las 5-tuplas [1,1,0,1,0] y [0,0,1,1,0].

Definamos en primer lugar las restricciones del problema. Llamaremos v_i al valor del i-ésimo elemento. En una etapa k cualquiera podemos considerar que una condición para que pueda encontrarse solución es que se cumpla que:

$$\sum_{i=1}^{k} v_i x_i + \sum_{i=k+1}^{n} v_i \geq M$$

es decir, en una etapa k cualquiera, la suma de todos los elementos que se han considerado hasta esa etapa más el valor de todos los que faltan por considerar al menos ha de ser igual al valor M dado, porque si no, si ni siquiera introduciendo todos se llega a alcanzar este valor, significa que por este camino no hay solución. Es más, como sabemos que los elementos están en orden no decreciente podemos afirmar que:

$$\sum_{i=1}^{k} v_i x_i + v_{k+1} \leq M \ ,$$

es decir, que si al valor conseguido hasta la etapa k le añadimos el siguiente elemento (que es el menor) y ya se supera el valor de M, esto significa que no será posible alcanzar la solución por este camino.

Estas dos restricciones nos van a permitir reducir la búsqueda al evitar caminos que no conducen a solución. En el algoritmo, llamaremos:

$$s = \sum_{j=1}^{k-1} v_j x_j \quad \text{y} \quad r = \sum_{j=k}^{n} v_j \,.$$

Con esto, el procedimiento que encuentra todos los posibles subconjuntos es el siguiente:

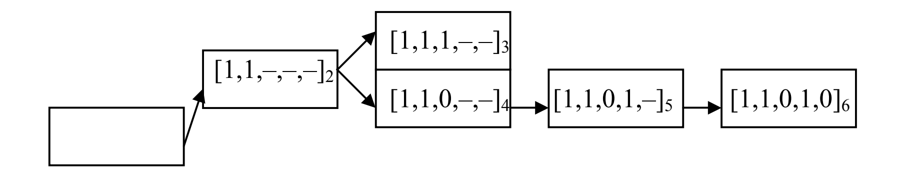
```
CONST n = ...; (* numero de elementos *)
      M = ...; (* cantidad dada *)

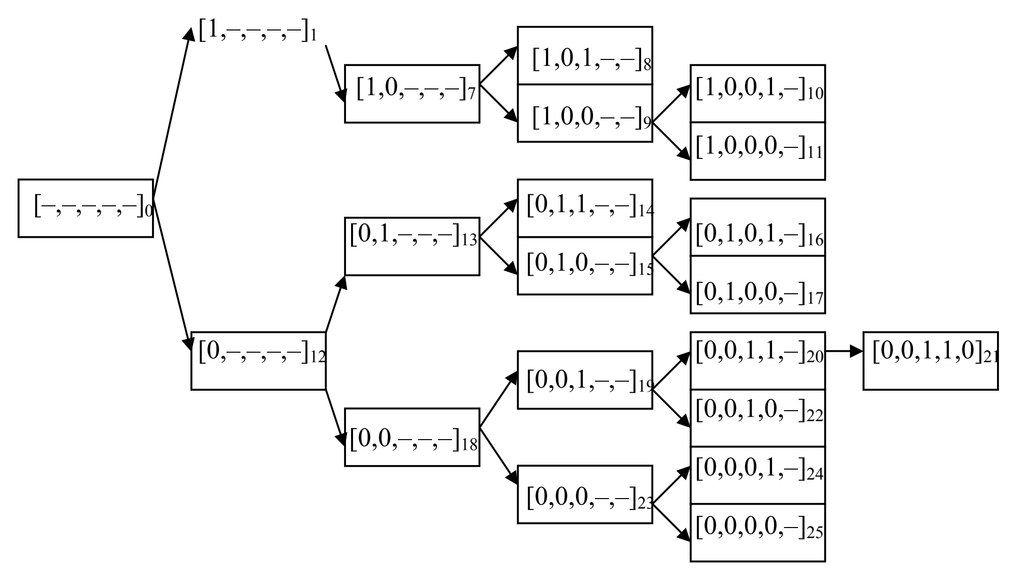
TYPE CONJUNTO = ARRAY [1..n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;

VAR  v:CONJUNTO;
     X:SOLUCION;

PROCEDURE Subconjuntos(s,k,r:CARDINAL);
BEGIN
   X[k]:=1;
   IF s+v[k]=M THEN
      ComunicarSolucion(k)
   ELSIF k=n THEN
      RETURN
   ELSIF s+v[k]+v[k+1]<=M THEN
      Subconjuntos(s+v[k],k+1,r-v[k])
   END;
   IF(s+r-v[k]>=M) AND(s+v[k+1]<=M) THEN
      X[k]:=0;
      Subconjuntos(s,k+1,r-v[k])
   END
END Subconjuntos;
```

Observemos cómo se va construyendo el árbol de expansión. En cada etapa k hemos de decidir, caso de que no se haya alcanzado la solución, si es posible añadir este elemento k al subconjunto y continuar por el hijo izquierdo, o no considerarlo y continuar por el derecho. Repitiendo el mismo proceso en cada etapa hasta o bien alcanzar la solución y en tal caso comunicarla o bien determinar la imposibilidad de continuar por esa rama. En ambos casos es necesario la vuelta atrás retrocediendo al nivel anterior. Para el ejemplo inicial del conjunto $W = \{2,3,5,10,20\}$ y $M = 15$, el árbol que va construyendo el algoritmo es:





cuyas soluciones son [1,1,0,1,0] y [0,0,1,1,0].

Para finalizar, el algoritmo ha de invocarse inicialmente desde el programa principal como *Subconjuntos*(s,1,r), donde $s = 0$ y r es igual a la suma de todos los elementos del conjunto.

6.10 CICLOS HAMILTONIANOS. EL VIAJANTE DE COMERCIO

Dado un grafo conexo, se llama *Ciclo Hamiltoniano* a aquel ciclo que visita exactamente una vez cada vértice del grafo y vuelve al punto de partida. El problema consiste en detectar la presencia de ciclos Hamiltonianos en un grafo dado.

Solución (👓)

Suponiendo como hemos hecho hasta ahora que los vértices del grafo están numerados desde 1 hasta n, la solución al problema puede expresarse como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$, donde x_i representa el i-ésimo vértice del ciclo Hamiltoniano. El algoritmo que resuelve el problema trabajará por etapas, decidiendo en cada etapa qué vértice del grafo de los aún no considerados puede formar parte del ciclo. Así, el algoritmo que resuelve el problema puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
     GRAFO  = ARRAY[1..n],[1..n] OF BOOLEAN;
VAR g:GRAFO; X:SOLUCION; existe:BOOLEAN;
PROCEDURE Hamiltoniano1(k:CARDINAL; VAR existe:BOOLEAN);
(* comprueba si existe un ciclo Hamiltoniano *)
BEGIN
```

```
   LOOP
      NuevoVertice(k);
      IF X[k] = 0 THEN EXIT END;
      IF k = n THEN existe:=TRUE
      ELSE Hamiltoniano1(k+1,existe)
      END
   END
END Hamiltoniano1;
```

Siendo el procedimiento *NuevoVertice* el que busca el siguiente vértice libre que pueda formar parte del ciclo y lo almacena en el vector solución *X*:

```
PROCEDURE NuevoVertice(k: CARDINAL);
   VAR j:CARDINAL; s:BOOLEAN;
BEGIN
   LOOP
      X[k]:=(X[k]+1) MOD (n+1);
      IF X[k]=0 THEN RETURN END;
      IF g[X[k-1],X[k]] THEN
         j:=1; s:=TRUE;
         WHILE (j<=k-1) AND s DO
            s:=(X[j]<>X[k]); INC(j)
         END;
         IF (j=k)AND((k<n)OR((k=n)AND(g[X[n],1]))) THEN RETURN END
      END
   END
END NuevoVertice;
```

El algoritmo termina cuando encuentra un ciclo o bien cuando ha analizado todas las posibilidades y no encuentra solución. Y respecto al estado de las variables y los parámetros de su invocación inicial, debe realizarse como sigue:

```
...
existe:=FALSE;
X[1]:=1;
Hamiltoniano1(2,existe);
IF existe THEN ComunicarSolucion(X) ELSE ...
...
```

Nos podemos plantear también una modificación al algoritmo para que encuentre todos los ciclos Hamiltonianos si es que hubiera más de uno. La modificación en este caso es simple:

```
PROCEDURE Hamiltoniano2(k:CARDINAL);
(* determina todos los ciclos *)
BEGIN
```

```
   LOOP
      NuevoVertice(k);
      IF X[k]=0 THEN RETURN END
      IF k=n THEN ComunicarSolucion(X)
      ELSE Hamiltoniano2(k+1)
      END
   END
END Hamiltoniano2;
```

Asimismo sería interesante generalizar el procedimiento anterior para tratar con grafos ponderados. El problema consistiría en diseñar un algoritmo que, dado un grafo ponderado con pesos positivos, encuentre el ciclo Hamiltoniano de coste mínimo.

En este caso hay que encontrar todos los ciclos Hamiltonianos, siendo necesario calcular el coste de cada solución y actualizar para obtener la óptima (el ciclo con mínimo coste). Lo interesante es que este problema coincide con nuestro viejo conocido el problema del viajante de comercio, ampliamente discutido en capítulos anteriores, y cuya solución mediante Vuelta Atrás es la que sigue:

```
PROCEDURE Hamiltoniano3(k:CARDINAL);
(* calcula el ciclo Hamiltoniano de minimo coste *)
BEGIN
   LOOP
      NuevoVertice2(k);
      IF X[k]=0 THEN RETURN END;
      IF k=n THEN
         coste:=CalcularCoste(X);
         IF coste<minimo THEN
            minimo:=coste;
            XMIN:=X
         END
      ELSE Hamiltoniano3(k+1)
      END
   END
END Hamiltoniano3;
```

La función *CalcularCoste* es la que obtiene la suma de los elementos de la *n*-tupla solución construida. Por otro lado, también se hace uso de dos variables globales para almacenar el coste mínimo y el camino:

```
VAR minimo:CARDINAL; XMIN:SOLUCION;
```

Por su parte, el procedimiento *NuevoVertice2* trabajará en este caso con un grafo ponderado, y por tanto los elementos de su matriz de adyacencia *g* serán enteros no negativos:

```
PROCEDURE NuevoVertice2(k:CARDINAL);
```

```
   VAR j:CARDINAL; s,vuelta:BOOLEAN;
BEGIN
   LOOP
      X[k]:=(X[k]+1) MOD (n+1);
      IF X[k]=0 THEN RETURN END;
      IF g[X[k-1],X[k]]<>MAX(CARDINAL) THEN
         j:=1; s:=TRUE;
         WHILE (j<=k-1) AND s DO
            s:=(X[j]<>X[k]); INC(j)
         END;
         vuelta:=g[X[n],1]<>MAX(CARDINAL);
         IF (j=k) AND ((k<n)OR((k=n)AND vuelta)) THEN RETURN END
      END
   END
END NuevoVertice2.
```

Este algoritmo debe ser invocado desde el programa principal tras inicializar la variable *minimo* y el primer elemento del vector *X* con el vértice desde donde parte el ciclo.

```
...
minimo:=MAX(CARDINAL);
X[1]:=1;
Hamiltoniano3(2);
IF minimo<MAX(CARDINAL) THEN ComunicarSolucion(XMIN)
...
```

6.11 EL CONTINENTAL

En el solitario de mesa llamado *Continental*, treinta y dos piezas se colocan en un tablero de treinta y tres casillas tal y como indica la siguiente figura, quedando vacía únicamente la casilla central:

```
        o  o  o
        o  o  o
  o  o  o  o  o  o  o
  o  o  o     o  o  o
  o  o  o  o  o  o  o
        o  o  o
        o  o  o
```

Una pieza sólo puede moverse saltando sobre una de sus vecinas y cayendo en una posición vacía, al igual que en el juego de las *damas*, aunque aquí no están permitidos los saltos en diagonal. La pieza sobre la que se salta se retira del tablero.

El problema consiste en diseñar un algoritmo que encuentre una serie de movimientos (saltos) que, partiendo de la configuración inicial expuesta en la figura, llegue a una situación en donde sólo quede una pieza en el tablero, que ha de estar en la posición central.

Solución (😐)

Representaremos el tablero como una matriz bidimensional en la que los elementos pueden tomar los valores *novalida*, *libre* u *ocupada*, dependiendo de que esa posición no sea válida (no correspoda a una de las posición que forman la "cruz"), o bien siendo válida exista ficha o no.

La solución vendrá dada en una tupla de valores $X = [x_1, x_2, ..., x_m]$ en donde x_i representa un movimiento (salto) del juego. En este valor se almacenará la posición de la ficha que va a efectuar el movimiento, la posición hacia donde da el salto y la posición de la ficha "comida". El valor *m* representa el número de saltos (movimientos) efectuados para alcanzar la solución. Puesto que en cada movimiento ha de comerse obligatoriamente una ficha, sabemos que *m* podrá tomar a lo sumo el valor 31. Con esto, el algoritmo que resuelve el problema es:

```
CONST m = 31;  (* numero maximo de movimientos *)
      n = 7;   (* tamano del tablero *)
TYPE ESTADO   = (libre,ocupada,novalida);(* tipo de casilla *)
     TABLERO  = ARRAY[1..n],[1..n] OF ESTADO;
     PAR      = RECORD x,y:INTEGER END; (* coordenadas *)
     SALTO    = RECORD origen,destino,comido:PAR END;
     SOLUCION = ARRAY [1..m] OF SALTO;

PROCEDURE Continental(VAR k:CARDINAL;VAR t:TABLERO;
                      VAR encontrado:BOOLEAN;VAR sol:SOLUCION);
   VAR i,j:CARDINAL;
BEGIN
   IF Fin(k,t) THEN encontrado:=TRUE
   ELSE
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            IF Valido(i,j,1,t,encontrado) THEN(* a la izquierda *)
               INC(k);
               sol[k].origen.x:=i;
               sol[k].origen.y:=j;
               sol[k].destino.x:=i;
               sol[k].destino.y:=j-2;
               sol[k].comido.x:=i;
               sol[k].comido.y:=j-1;
               NuevaTabla(t,i,j,1); (* actualiza el tablero *)
               Continental(k,t,encontrado,sol)
            END;
            IF Valido(i,j,2,t,encontrado) THEN (* hacia arriba *)
```

```
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i-2;
            sol[k].destino.y:=j;
            sol[k].comido.x:=i-1;
            sol[k].comido.y:=j;
            NuevaTabla(t,i,j,2);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
          END;
          IF Valido(i,j,3,t,encontrado) THEN (* a la derecha *)
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i;
            sol[k].destino.y:=j+2;
            sol[k].comido.x:=i;
            sol[k].comido.y:=j+1;
            NuevaTabla(t,i,j,3);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
          END;
          IF Valido(i,j,4,t,encontrado) THEN (* hacia abajo *)
            INC(k);
            sol[k].origen.x:=i;
            sol[k].origen.y:=j;
            sol[k].destino.x:=i+2;
            sol[k].destino.y:=j;
            sol[k].comido.x:=i+1;
            sol[k].comido.y:=j;
            NuevaTabla(t,i,j,4);(* actualiza el tablero *)
            Continental(k,t,encontrado,sol)
          END;
        END
      END;
      IF NOT encontrado THEN (* cancelar anotacion *)
        RestaurarTabla(t,k,sol);
        AnularSalida(sol,k);
        DEC(k)
      END
    END
  END Continental;
```

La función *Fin* determina si se ha llegado al final del juego, esto es, si sólo queda una ficha y ésta se encuentra en el centro del tablero, y la función *Valido* comprueba si una ficha puede moverse o no:

```
PROCEDURE Valido(i,j,mov:CARDINAL;VAR t:TABLERO;e:BOOLEAN):BOOLEAN;
BEGIN
   IF mov=1 THEN (* izquierda *)
      RETURN ((j-1>0) AND (t[i,j]=ocupada) AND(t[i,j-1]=ocupada)
              AND (j-2>0) AND (t[i,j-2]=libre) AND (NOT e))
   ELSIF mov=2 THEN (* arriba *)
      RETURN ((i-1>0) AND (t[i-1,j]=ocupada) AND(t[i,j]=ocupada)
              AND (i-2>0) AND (t[i-2,j]=libre) AND (NOT e))
   ELSIF mov=3 THEN (* derecha *)
      RETURN ((j+1<8) AND (t[i,j+1]=ocupada) AND(t[i,j]=ocupada)
              AND (j+2<8) AND (t[i,j+2]=libre) AND (NOT e))
   ELSIF mov=4 THEN (* abajo *)
      RETURN ((i+1<8) AND (t[i+1,j]=ocupada) AND(t[i,j]=ocupada)
              AND (i+2<8) AND (t[i+2,j]=libre) AND (NOT e))
   END
END Valido;
```

Un aspecto interesante de este problema es que pone de manifiesto la importancia que tiene el orden de generación de los nodos del árbol de expansión. Si en vez de seguir la secuencia de búsqueda utilizada en la anterior implementación (izquierda, arriba, derecha y abajo) se intentan los movimientos en otro orden, el tiempo de ejecución del algoritmo pasa de varios segundos a más de dos horas.

6.12 HORARIOS DE TRENES

El problema de los horarios de los trenes fue enunciado en el capítulo anterior (apartado 5.15). Una compañía de ferrocarriles que sirve n estaciones $S_1,...,S_n$ trata de mejorar su servicio al cliente mediante terminales de información. Dadas una estación origen S_o y una estación destino S_d, un terminal debe ofrecer (inmediatamente) la información sobre el horario de los trenes que hacen la conexión entre S_o y S_d y que minimizan el tiempo de trayecto total.

Necesitamos por tanto implementar un algoritmo que realice esta tarea a partir de la tabla con los horarios, suponiendo que las horas de salida de los trenes coinciden con las de sus llegadas (es decir, que no hay tiempos de espera) y que, naturalmente, no todas las estaciones están conectadas entre sí por líneas directas; así, en muchos casos hay que hacer transbordos aunque se supone que tardan tiempo cero en efectuarse. Nos planteamos en este apartado solucionarlo mediante un algoritmo de Vuelta Atrás.

Solución (☺)

En primer lugar es necesario expresar la solución del problema mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$. En este caso, cada x_i va a representar el

número de estación que compone el trayecto más corto. La n-tupla estará rellena hasta la posición k-ésima, siendo k el número de estaciones que debe recorrer, dándose además que $x_1 = S_o$ y $x_k = S_d$. Por comodidad y sin ninguna pérdida de generalidad supondremos que las estaciones están numeradas del 1 al n.

En cada etapa iremos probando estaciones, con la restricción de que no pasemos dos veces por la misma y que la que añadamos nueva en cada paso esté conectada con la anterior.

Como además se trata de un problema de optimización utilizaremos una restricción adicional, como es la comprobación de que el tiempo acumulado hasta el momento por una solución en proceso no supere el tiempo alcanzado por una solución ya conocida. Estas ideas dan lugar al siguiente algoritmo:

```
CONST n = ...; (* numero de estaciones *)

TYPE SOLUCION = ARRAY [1..n] OF CARDINAL;
     HORARIOS = ARRAY [1..n],[1..n] OF CARDINAL;

VAR estacionorigen,estaciondestino,minimo:CARDINAL;
    tablatiempos:HORARIOS;
    X,solucionoptima:SOLUCION;

PROCEDURE Estaciones(k:CARDINAL;tiempoacum:CARDINAL);
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         tiempoacum:=tiempoacum+tablatiempos[X[k-1],X[k]];
         IF tiempoacum<=minimo THEN
            IF X[k]=estaciondestino THEN (* hemos llegado *)
               solucionoptima:=X; minimo:=tiempoacum
            ELSIF k<n THEN
               Estaciones(k+1,tiempoacum)
            END
         END
      END
   UNTIL X[k]=n
END Estaciones;
```

Las variables *estacionorigen* y *estaciondestino* son las que el usuario elige, y la matriz *tablatiempos* determina el tiempo de conexión entre cada par de estaciones, puediendo ser *tablatiempos*$[i,j] = \infty$ si no existe conexión entre las estaciones i y j. Por su parte, la función *Aceptable* comprueba las restricciones definidas anteriormente para el problema:

```
PROCEDURE Aceptable(k:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
```

```
BEGIN
   FOR i:=1 TO k-1 DO (* no puede haber estaciones repetidas *)
      IF X[i]=X[k] THEN RETURN FALSE END
   END;
   (* la nueva estacion ha de ser alcanzable desde la anterior *)
   RETURN tablatiempos[X[k-1],X[k]]<MAX(CARDINAL);
END Aceptable;
```

Obsérvese cómo en este caso la solución puede estar compuesta por menos de n valores significativos. De esta forma, de la n-tupla solución X construida sólo hemos de considerar los k primeros elementos, siendo k el primer índice para el que $X[k]$ = *estaciondestino*.

Por otro lado, el programa principal, además de pedir al usuario las estaciones origen y destino y de dar valores a la tabla de tiempos entre estaciones, debe inicializar la variable *minimo* y el vector con la estacion origen, e invocar al procedimiento que realiza el proceso de Vuelta Atrás:

```
...
minimo:=MAX(CARDINAL);
X[1]:=estacionorigen;
Estaciones(2,0);
IF minimo<MAX(CARDINAL) THEN ComunicarSolucion(solucion)
ELSE ...
```

Al final, la variable global *solucion* contendrá el recorrido óptimo y la variable *minimo* el tiempo total de ese trayecto, a menos que el problema no tenga solución (como ocurre por ejemplo en el caso de que la estación destino no sea alcanzable desde la estación origen) en cuyo caso la variable *minimo* seguirá valiendo ∞.

6.13 LA ASIGNACIÓN DE TAREAS EN PARALELO

Supongamos que necesitamos realizar n tareas en una máquina multiprocesador, pero que sólo m procesadores pueden trabajar en paralelo. Sea t_i el tiempo de ejecución de la i-ésima tarea ($1 \le i \le n$).

El problema consiste en implementar un algoritmo que determine en qué procesador y en qué orden hay que ejecutar los trabajos, de forma que el tiempo total de ejecución sea mínimo.

Solución (☺)

Para resolver este problema expresaremos su solución mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde cada valor x_i representa el procesador que realiza la i-ésima tarea. Minimizar el tiempo total de ejecución significará encontrar la solución que tenga menor el máximo de los tiempos de cada procesador.

Las restricciones explícitas establecen el rango de valores que pueden tomar los componentes x_i de la n-tupla solución, que en este caso han de ser números comprendidos entre 1 y m.

Por otro lado no definiremos ninguna restricción implícita puesto que a priori cualquier *n*-tupla formada por valores que cumplan las restricciones explícitas es suceptible de ser solución. Con todo esto, el algoritmo que resuelve el problema puede ser planteado como sigue:

```
CONST m = ...; (* numero de procesadores *)
      n = ...; (* numero de tareas *)
TYPE TIEMPOS  = ARRAY [1..n] OF CARDINAL;
     SOLUCION = ARRAY [1..n] OF CARDINAL;
     VALOR    = ARRAY [1..m] OF CARDINAL;
VAR tiempos:TIEMPOS; (* tiempo de ejecucion de cada tarea *)
    mejor: CARDINAL;
    X,solucion:SOLUCION;
    valor:VALOR; (* tiempo empleado por cada procesador *)

PROCEDURE Procesador(k:CARDINAL);
   VAR t,j:CARDINAL;
BEGIN
   FOR j:=1 TO m DO (* probamos con todos los procesadores *)
      X[k]:=j; valor[j]:=valor[j]+tiempos[k];
      IF k<n THEN Procesador(k+1);
      ELSE
         t:=CalcularTiempo();
         IF mejor>t THEN mejor:=t; solucion:=X END
      END;
      valor[j]:=valor[j]-tiempos[k];
   END
END Procesador;
```

La función *CalcularTiempo* es la que calcula el tiempo total máximo para una tupla solución:

```
PROCEDURE CalcularTiempo():CARDINAL;
   VAR i,maximo: CARDINAL;
BEGIN
   maximo:=valor[1];
   FOR i:=2 TO m DO
      IF valor[i]>maximo THEN maximo:=valor[i] END
   END;
   RETURN maximo
END CalcularTiempo;
```

El programa principal ha de inicializar la variable *mejor* y ha de invocar al procedimiento *Procesador*:

```
...
mejor:=MAX(CARDINAL);
```

```
Procesador(1);
ComunicarSolucion(solucion);
...
```

6.14 EL COLOREADO DE MAPAS

Dado un grafo conexo y un número $m > 0$, llamamos *colorear* el grafo a asignar un número i ($1 \le i \le m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales. Deseamos implementar un algoritmo que coloree un grafo dado.

Solución (☺)

El nombre de este problema proviene de un problema clásico, el del coloreado de mapas en el plano. Para resolverlo se utilizan grafos puesto que un mapa puede ser representado por un grafo conexo. Cada vértice corresponde a un país y cada arco entre dos vértices indica que los dos países son vecinos. Desde el siglo XVII ya se conoce que con cuatro colores basta para colorear cualquier mapa planar, pero sin embargo existen situaciones en donde no nos importa el número de colores que se utilicen.

Para implementar un algoritmo de Vuelta Atrás, la solución al problema puede expresarse como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde x_i representa el color del i-ésimo vértice. El algoritmo que resuelve el problema trabajará por etapas, asignando en cada etapa k un color (entre 1 y m) al vértice k-ésimo.

En primer lugar, y para un grafo con n vértices y con m colores, el algoritmo que encuentra una solución al problema es el siguiente:

```
CONST n = ...; (* numero de vertices *)
      m = ...; (* numero maximo de colores  *)

TYPE  GRAFO=ARRAY[1..n],[1..n]OF BOOLEAN; (* matriz adyacencia *)
      SOLUCION = ARRAY [1..n] OF CARDINAL;

VAR   g:GRAFO; X:SOLUCION; exito:BOOLEAN

PROCEDURE Colorear1(k:CARDINAL); (* busca una posible solucion *)
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear1(k+1)
         ELSE exito:=TRUE
         END
      END
   UNTIL (exito) OR (X[k]=m)
```

```
END Colorear1;
```

La función *Aceptable* es la que comprueba las restricción definida para este problema, que consiste en que dos países vecinos (vértices adyacentes) no pueden tener el mismo color:

```
PROCEDURE Aceptable(k: CARDINAL): BOOLEAN;
   VAR j: CARDINAL;
BEGIN
   FOR j:=1 TO k-1 DO
      IF (g[k,j]) AND (X[k]=X[j]) THEN RETURN FALSE END
   END;
   RETURN TRUE
END Aceptable;
```

El programa principal del algoritmo ha de invocar a la función *Colorear1* como sigue:

```
...
exito:=FALSE;
Colorear1(1);
IF exito THEN ComunicarSolucion(X)
...
```

Supongamos ahora que lo que deseamos es obtener todas las formas distintas de colorear un grafo. Entonces el algoritmo anterior podría ser modificado de la siguiente manera:

```
PROCEDURE Colorear2(k:CARDINAL);
(* busca todas las soluciones *)
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear2(k+1)
         ELSE ComunicarSolucion(X)
         END
      END
   UNTIL (X[k]=m)
END Colorear2;
```

Por último, vamos a mostrar el algoritmo para colorear un grafo con el mínimo número de colores:

```
PROCEDURE Colorear3(k:CARDINAL);
(* busca la solucion optima *)
   VAR numcolores:CARDINAL;
```

```
BEGIN
   X[k]:=0;
   REPEAT
      INC(X[k]);
      IF Aceptable(k) THEN
         IF k<n THEN Colorear3(k+1)
         ELSE
            numcolores:=NumeroColores(X);
            IF minimo>numcolores THEN
               mejor:=X;
               minimo:=numcolores
            END
         END
      END
   UNTIL (X[k]=m)
END Colorear3;
```

La función *NumeroColores* es la que calcula el número de colores utilizado en una solución. Por la forma en la que hemos ido construyendo las soluciones no queda garantizado que los colores utilizados posean números consecutivos, de forma que es necesario comprobar todos los colores para saber cuales han sido usados en una solución concreta:

```
PROCEDURE NumeroColores(X:SOLUCION):CARDINAL;
   VAR i,j,suma:CARDINAL; sigo:BOOLEAN;
BEGIN
   suma:=0;
   FOR j:=1 TO m DO (* recorremos todos los colores *)
      i:=1;
      sigo:=FALSE;
      WHILE (i<n) AND sigo DO
         IF X[i]=j THEN (* encontrado el color j *)
            INC(suma);
            sigo:=FALSE
         END
      END
   END;
   RETURN suma;
END NumeroColores;
```

En estos algoritmos es importante hacer notar que la constante *m* que indica el número máximo de colores a utilizar ha de ser mayor o igual a cuatro, pues se sabe que con cuatro colores basta siempre que el grafo corresponda a un mapa. Ahora bien, conviene también observar que no todo grafo conexo representa a un mapa planar; por ejemplo un grafo de cinco vértices completamente conexo, es decir, que tenga todos sus vértices conectados entre sí, no puede corresponder a un mapa en el plano.

6.15 RECONOCIMIENTO DE GRAFOS

Dadas dos matrices de adyacencia, el problema consiste en determinar si ambas representan al mismo grafo, salvo nombres de los vértices.

Solución (☞)

Nuestro punto de partida son dos matrices cuadradas L_1 y L_2 que representan las matrices de adyacencia de dos grafos g_1 y g_2. Queremos ver si g_1 y g_2 son iguales, salvo por la numeración en sus vértices.

Podemos suponer sin pérdida de generalidad que la dimensión de ambas matrices coincide. Si no, al estar suponiendo que los vértices de los grafos están numerados de forma consecutiva a partir de 1, ya podríamos decidir que ambos grafos son distintos por tener diferente número de vértices. Sea entonces n la dimensión de ambas matrices.

Supondremos además, por ser el caso más general, que los grafos son ponderados y dirigidos, y por ello definimos el tipo de las matrices de adyacencia como sigue:

```
CONST n = ...; (* numero de vertices *)
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
```

Para resolver este problema lo que necesitamos es realizar una aplicación, si es posible, entre los vértices del primer grafo y los del segundo. Por tanto, podemos representar la solución como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$, donde x_i va a indicar el vértice de g_2 que corresponde al i-ésimo vértice de g_1.

En cada etapa el algoritmo, empezando por el vértice 1 de g_1, va a ir construyendo la solución, tratando de asignar un vértice de g_2 a cada uno de g_1.

Una vez tenemos expresada de esta forma la solución podemos definir las restricciones que aplicaremos al problema, puesto que si no el árbol de expansión constaría de n^n nodos, demasiados para que el algoritmo sea operativo.

En primer lugar, no se pueden repetir vértices, es decir, los elementos de la n-tupla X han de ser todos distintos (una permutación de los n vértices de g_2). Además, el vértice de g_2 indicado por x_k ha de tener el mismo número de arcos (entrantes y salientes de él) que el correspondiente vértice k de g_1.

Por otro lado, para que al añadir el elemento k-ésimo a una solución parcial sea k-prometedora, las conexiones (arcos) entre el nuevo elemento x_k y los ya asignados en la solución parcial X ha de coincidir con las existentes en los correspondientes vértices de g_1. Esto hace que el algoritmo que resuelve el problema pueda ser implementado como sigue:

```
PROCEDURE GrafosIguales(k:CARDINAL);
   VAR vertice:CARDINAL; (*vertice que indica la opcion en curso*)
BEGIN
   vertice:=0;
```

```
    REPEAT
       INC(vertice);
       X[k]:=vertice;
       IF Valido(k) THEN
          IF k<n THEN
             GrafosIguales(k+1)
          ELSE
             exito:=TRUE
          END
       END
    UNTIL (X[k]=n) OR exito;
 END GrafosIguales;
```

La "inteligencia" del algoritmo la suministra la función *Valido*, que es la que comprueba las restricciones anteriormente citadas:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO (* no pueden repetirse elementos *)
      IF X[i]=X[k] THEN
         RETURN FALSE
      END
   END;
   IF NumArcos(L1,k)<>NumArcos(L2,X[k]) THEN(* mismo num. arcos *)
      RETURN FALSE
   END;
   FOR i:=1 TO k-1 DO (* mismas conexiones *)
      IF (L2[X[i],X[k]]<>L1[i,k]) OR (L2[X[k],X[i]]<>L1[k,i]) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE;
END Valido;
```

La función *NumArcos* es la que, dada una matriz de adyacencia de un grafo dirigido y ponderado como los que estamos considerando, y uno de sus vértices, devuelve el número de arcos que salen y entran de él:

```
PROCEDURE NumArcos(VAR L:MATRIZ;k:CARDINAL):CARDINAL;
   VAR i,suma:CARDINAL;
BEGIN
   suma:=0;
   FOR i:=1 TO n DO
      IF ((i<>k) AND (L[i,k]<MAX(CARDINAL))) THEN
         INC(suma)
```

```
      END;
      IF ((i<>k) AND (L[k,i]<MAX(CARDINAL))) THEN
         INC(suma)
      END;
   END;
   RETURN suma;
END NumArcos;
```

6.16 SUBCONJUNTOS DE IGUAL SUMA

Dado un conjunto de n enteros, necesitamos decidir si puede ser descompuesto en dos subconjuntos disjuntos cuyos elementos sumen la misma cantidad.

Solución (☺)

Para resolver el problema almacenaremos el conjunto en un vector de enteros y representaremos la solución mediante una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde cada componente x_i puede tomar los valores 1 ó 2 indicando que el i-ésimo elemento pertenece al subconjunto 1 o al subconjunto 2 respectivamente.

En cuanto a las restricciones implícitas, la primera condición que exigiremos al conjunto de enteros para que pueda ser descompuesto en dos subconjuntos que sumen igual es que la suma de los elementos del conjunto sea un número par. Esto lo comprobaremos en el programa principal, y antes de invocar por primera vez al procedimiento recursivo que realiza el algoritmo Vuelta Atrás.

En cada etapa k intentaremos colocar el k-ésimo elemento del conjunto en uno de los dos subconjuntos posibles, lo que da lugar al siguiente procedimiento:

```
CONST n = ...; (* numero de elementos del conjunto *);

TYPE CONJUNTO = ARRAY[1..n] OF INTEGER;
     SOLUCION = ARRAY[1..n] OF CARDINAL;

VAR X:SOLUCION; numeros:CONJUNTO;
    suma:ARRAY[1..2] OF INTEGER; (* suma acumulada de
                                    los subconjuntos *)


PROCEDURE DosSubconjuntos(k:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=1 TO 2 DO (* cada una de las dos posibilidades *)
      X[k]:=j;
      suma[j]:=suma[j]+numeros[k];
      IF k<n THEN
```

```
        DosSubconjuntos(k+1)
      ELSIF suma[1]=suma[2] THEN
        ComunicarSolucion;
      END;
      suma[j]:=suma[j]-numeros[k] (* cancelar anotacion *)
    END
  END DosSubconjuntos;
```

En este problema podemos también definir una restricción en forma de cota que permita realizar la poda de aquellos nodos del árbol de expansión que sepamos que no conducen a una solución. La idea consiste en sumar al principio todos los elementos del conjunto, que en cualquier caso hemos de hacer para ver si es un número par. Con esta suma, que almacenaremos en la variable global *sumatotal*, podemos dejar de explorar aquellos nodos del árbol de expansión que verifiquen que la suma de uno de los dos subconjuntos que están construyendo sea mayor que la mitad de la suma total:

```
PROCEDURE DosSubconjuntos2(k:CARDINAL);
  VAR j:CARDINAL;
BEGIN
  FOR j:=1 TO 2 DO
    X[k]:=j;
    suma[j]:=suma[j]+numeros[k];
    IF suma[j]<=(sumatotal DIV 2) THEN (* poda *)
      IF k<n THEN DosSubconjuntos2(k+1)
      ELSIF suma[1]=suma[2] THEN ComunicarSolucion;
      END
    END;
    suma[j]:=suma[j]-numeros[k] (* cancelar anotacion *)
  END
END DosSubconjuntos2;
```

De esta forma conseguimos incrementar las restricciones del problema, lo que contribuye a una menor expansión del número de nodos y por tanto a una mayor eficiencia del algoritmo resultante.

6.17 LAS MÚLTIPLES MOCHILAS (0,1)

Dados n elementos, cada uno con un beneficio b_i y un peso p_i asociado ($1 \le i \le n$), y m mochilas, cada una con una capacidad k_j ($1 \le j \le m$), el problema de las Múltiples Mochilas (0,1) puede describirse como la asignación de los elementos a las mochilas de forma que se maximice el beneficio total de todos los elementos asignados sin superar la capacidad de las mochilas, y teniendo en cuenta que cada elemento puede ser asignado a una mochila o a ninguna, y que un elemento aportará beneficio sólo si éste es introducido en una mochila.

Nos planteamos diseñar un algoritmo Vuelta Atrás para la resolución del problema de las Múltiples Mochilas (0,1) para cualquier conjunto de elementos y mochilas.

Solución (☺)

La solución al problema puede expresarse como una n-tupla $X = [x_1, x_2, ..., x_n]$ en donde x_i representa la mochila en donde es introducido el i-ésimo elemento, o bien x_i puede valer cero si el elemento no se introduce en ninguna mochila. En cada etapa k el algoritmo irá construyendo la tupla solución, intentando decidir en qué mochila introduce el k-ésimo elemento, o si lo deja fuera.

En este caso las restricciones consisten en comprobar que no se supera la capacidad de la mochila indicada. Esto da lugar al siguiente programa:

```
CONST n = ...; (* numero de elementos *)
      m = ...; (* numero de mochilas *)
TYPE MOCHILAS = ARRAY[1..m] OF INTEGER;
     SOLUCION = ARRAY[1..n] OF INTEGER
     PAR = RECORD peso,beneficio:INTEGER END;
     ELEMENTOS = ARRAY[1..n] OF PAR;
VAR cap:ARRAY[1..m]OF CARDINAL;(*capacidad libre de las mochilas *)
      ben,benoptimo:CARDINAL;
      X,soloptima:SOLUCION;
      elem:ELEMENTOS;

PROCEDURE MochilaMultiple(k:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=0 TO m DO
      IF (j=0) OR (cap[j]>=elem[k].peso) THEN (* restricciones *)
         X[k]:=j;
         IF j>0 THEN (* hacer anotacion *)
            cap[j]:=cap[j]-elem[k].peso;
            ben:=ben+elem[k].beneficio
         END;
         IF k<n THEN MochilaMultiple(k+1)
         ELSIF ben>benoptimo THEN
            benoptimo:=ben; soloptima:=X (* actualizar solucion *)
         END;
         IF j>0 THEN (* cancelar anotacion *)
            cap[j]:=cap[j]+elem[k].peso;
            ben:=ben-elem[k].beneficio
         END
      END
   END
END MochilaMultiple;
```

Como puede observarse, este ejemplo es una muestra clara de las ventajas que ofrece el disponer de un esquema general de diseño de algoritmos Vuelta Atrás, pues permite resolver los problemas de forma sencilla, unificada y general.